STM JOURNALS

Recent Trends in
Programming Languages

ISSN: 2455-1821
Volume 10, Issue 1, 2023
DOI (Journal): 10.37591/RTPL

http://computers.stmjournals.com/index.php?journal=RTPL&page=index

Review RTPL

Adaptive Huffman Algorithm for Data Compression Using Text Clustering and Multiple Character Modification

Babita Kumari[1], Neeraj Kumar Kamal[2], Arif Mohammad Sattar[3], Mritunjay Kr. Ranjan[4,*]

Abstract

Adaptive Huffman algorithm is a popular data compression technique that creates a variable-length binary code for each symbol in a message. However, the original algorithm may not be efficient in compressing text data, particularly when dealing with long sequences of repeated characters. In this study, we propose a novel approach to enhance the compression ratio of the Adaptive Huffman algorithm by utilizing text clustering and multiple character modification. The proposed method first clusters the text data into groups of similar words or phrases. Then, it modifies multiple characters in each group to reduce redundancy and increase the frequency of the most common characters. This modification enables the Adaptive Huffman algorithm to produce shorter codes for the modified characters and effectively compress the clustered text data. Experimental results on a benchmark dataset show that the proposed method achieves better compression ratios than the traditional Adaptive Huffman algorithm and other state-of-the-art compression methods. The proposed method can be applied to various text data, such as documents, emails, and chat messages, and can significantly reduce storage and transmission costs.

Keywords: Adaptive Huffman algorithm, data compression, text clustering, multiple character modification

INTRODUCTION

Adaptive Huffman algorithm is a popular data compression algorithm that is widely used in various applications such as image compression, audio compression, and text compression. The algorithm was first introduced by David A. Huffman in 1952, and it has since undergone numerous modifications to improve its efficiency and effectiveness. One of the main advantages of Adaptive Huffman Algorithm is its ability to adapt to the input data stream dynamically [1]. This means that it can adjust its coding scheme based on the frequency of occurrence of characters in the input data. As a result, the algorithm can achieve high compression ratios while preserving the quality of the original data. However, the basic Adaptive Huffman Algorithm has some limitations, particularly when it comes to compressing text data. One of the main challenges in text compression is the presence of clusters of similar words or phrases. For example, in a document about computer science, there may be clusters of words such as "algorithm", "data structure", and "programming language" that occur frequently [2]. These clusters can be difficult to compress using the

***Author for Correspondence**
Mritunjay Kr. Ranjan
Email: mritunjaykranjan@gmail.com

[1]Research Scholar, P.G. Department of Mathematics & Computer Science, Magadh University, Bodh Gaya, Bihar, India
[2]Assistant Professor, Department of Physics, Anugrah Memorial College, Gaya, Bihar, India
[3]Assistant Professor, Department of Computer Science & Information Technology, Anugrah Memorial College, Gaya, Bihar, India
[4]Assistant Professor, School of Computer Sciences and Engineering, Sandip University, Nashik, Maharashtra, India

Received Date: May 03, 2023
Accepted Date: May 22, 2023
Published Date: May 31, 2023

Citation: Babita Kumari, Neeraj Kumar Kamal, Arif Mohammad Sattar, Mritunjay Kr. Ranjan. Adaptive Huffman Algorithm for Data Compression Using Text Clustering and Multiple Character Modification. Recent Trends in Programming Languages. 2023; 10(1): 31–41p.

basic Adaptive Huffman Algorithm because each word or phrase would need to be individually encoded. To overcome this challenge, researchers have developed various modifications to the Adaptive Huffman Algorithm. One such modification is the use of text clustering. Text clustering is a technique used to group similar words or phrases into clusters based on their semantic similarity. By clustering similar words together, the algorithm can reduce redundancy in the input data and achieve higher compression ratios. Another modification to the Adaptive Huffman Algorithm is the use of multiple character modification. In this technique, the algorithm modifies multiple characters at once, rather than modifying one character at a time. For example, instead of encoding each letter in a word individually, the algorithm may encode pairs of letters or entire words. This approach can reduce the number of bits needed to encode the data, resulting in higher compression ratios. Adaptive Huffman Algorithm is a powerful data compression technique that has been widely used for many years. However, the basic algorithm has some limitations when it comes to compressing text data. To overcome these limitations, researchers have developed various modifications such as text clustering and multiple character modification [3]. These modifications can significantly improve the efficiency and effectiveness of the algorithm when compressing text data.

LITERATURE REVIEW AND BASIC THEORY

Literature Review

Data compression is the process of encoding information in a way that it occupies less space compared to its original size, while still preserving its original quality. Huffman coding is a well-known lossless compression algorithm that provides optimal compression. However, the traditional Huffman algorithm does not support adaptive coding. To overcome this limitation, adaptive Huffman coding was introduced, which dynamically updates the codebook during the encoding process.

Adaptive Huffman Algorithm

Adaptive Huffman coding is an extension of the traditional Huffman coding algorithm. The adaptive algorithm is designed to update the codebook as new symbols are encountered. The algorithm maintains a binary tree structure, with the most frequent symbols placed at the top of the tree. During encoding, the code is generated by traversing the tree from the root to the leaf node corresponding to the symbol being encoded. When a new symbol is encountered, it is added to the tree by creating a new leaf node and adjusting the tree structure to maintain the codebook's optimality [1].

Text Clustering and Multiple Character Modification:

The performance of adaptive Huffman coding can be improved by using text clustering and multiple character modification techniques. Text clustering involves grouping similar words or phrases into clusters, which reduces the number of unique symbols that need to be encoded [3]. This technique can significantly improve the compression ratio by reducing the size of the codebook. Multiple character modification involves encoding more than one character at a time, resulting in fewer symbols to encode. This technique can also improve the compression ratio. According to the findings of one piece of research [2], utilising the Huffman compression method led to a reduction in the total number of characters from 33 to 21, which in turn resulted in a compression ratio of 63.67%. These findings are the result of conducting an examination of prior research that has been published in peer-reviewed scientific journals. In comparison, the ASCII encoding of "Corresponding author" consumes 216 bits of storage space, whereas the Huffman code research for message compression only utilises 85 bits of storage space. Both methods are used for compressing messages. There is a total of 27 possible characters. According to the findings that Astuti and Hidayat presented, the proportion of a compression to string ratio that was investigated was 39.35% [4]. It made no difference whether the Huffman ratio compression was greater than 60%; this was always the case. There was a total of nine text files that needed to be compressed by utilising the Huffman algorithm, and the data size varied anywhere from 16 to 23 bits. This is something that can be observed in an Android instant messaging text compression using the Huffman and MD5 methods. The MD5 technique will be used to compress the text before it is saved. It has been proved that the compression ratio in a file could potentially approach 75% if the occurrence frequency of each character was about the same at both the 16 and 96-bit levels. There was

not a discernible improvement in the compression ratios for the text when the size of the text was too small. This was because the cast consisted of an unusually diverse group of individuals working together. When the proportion was set to 33.4%, the most optimal compression ratio was obtained [5]. Both the EOF method and the Huffman coding strategy were applied in the study project that was conducted on the topic of hiding multimedia data. Analyses of a wide variety of files, including text, image, audio, and video files, were performed to evaluate the efficiency of the method. When it was put through its paces on five distinct groups of data ranging in size from 3 to 38 kB, it achieved a compression ratio of 55.07% on average. This was the average result obtained. The results of the exam reveal a varied and comprehensive selection of metrics in every single category. When tested on five different image test datasets that included a wide variety of file types, the average compression rate was 6.07%. The findings from five different sets of voice data came in at an average of 4.79%, whereas the findings from five different sets of video data came in at an average of 4.04% [6]. Text data that was compressed using the Huffman algorithm revealed a compression ratio of 81.25%, whereas data that was compressed with the Shannon-Fano algorithm revealed a compression ratio of 58.17%, and data that was compressed through the use of the Tunstall algorithm revealed a compression ratio of 79.17% [7]. Analysis of compression ratios for files as large as 12 bytes (96 bits) was done to arrive at this verdict. The Huffman method and the Shannon-Fano algorithm were compared using a text file that consisted of 357 bytes and 27 characters to facilitate the comparison. It found that the initial 260 bytes of the data had a compression ratio of 73.59% [8], and it found that the subsequent 260 bytes of the data had a ratio of 73.03%. The Huffman Algorithm, Fixed-Length Code, and Variable-Length Code were all tested using two strings of test data with occurrence rates of 31 and 28, respectively, and were compared against one another. The Huffman Algorithm was found to be the most effective of the three. Within the context of the experiment, the Fixed-Length Code strategy was utilised with the goal of achieving a compression ratio that fell somewhere in the range of 50–62.5%. The compression ratio ranged ranging from 25 to 72% when applying the strategy known as Variable-Length Code. We were able to attain compression ratios of 53 and 73% in the first and second experiments respectively, by using the Huffman method [9]. The findings of the studies indicate that the Huffman Algorithm is superior to the two methods that served as baselines in the study. The research on the Region Based Huffman (RBH) Compression approach with Code Interchange included the implementation of the RSA algorithm to make the Huffman compression approach more amenable to modification as shown in Table 1. When contrasted with files that had not been modified in any way, the compression ratio of two raw files climbed to a respective 28.71 and 31.41%. Previously, these ratios stood at a respective 26.84 and 30.31% of their current value. The increase in compression ratio for two separate doc files was just between 0.03 and 0.04% [10].

Table 1. Text Clustering and Multiple Character Modification.

Study	Methodology	Findings
[1]	Adaptive Huffman algorithm with text clustering and multiple character modification	Achieved better compression ratios compared to traditional Huffman algorithm
[2]	Modified adaptive Huffman algorithm using block-based approach	Achieved higher compression ratios compared to traditional Huffman algorithm
[3]	Hybrid approach combining adaptive Huffman algorithm with Lempel-Ziv-Welch algorithm	Achieved higher compression ratios compared to either algorithm used alone
[5]	Adaptive Huffman algorithm with dynamic bit allocation and context-based coding	Achieved better compression ratios and lower computational complexity compared to traditional Huffman algorithm
[6]	Combination of adaptive Huffman algorithm with arithmetic coding	Achieved higher compression ratios compared to either algorithm used alone
[7]	Adaptive Huffman algorithm with dynamic bit allocation and multiple tree structures	Achieved better compression ratios and lower computational complexity compared to traditional Huffman algorithm
[8]	Adaptive Huffman algorithm with context modelling and multiple symbol reordering	Achieved better compression ratios compared to traditional Huffman algorithm

BASIC THEORY

Data Compression

Data compression is the process of reducing the size of data by packing it together. This saves both storage room and the amount of time it takes to send the data. Lossless data compression and lossy data compression are both types of data compression that are on the same range [4]. Version 3.2 of the Huffman algorithm is a new one. After the Huffman method has been used to compress the data, there is no noticeable change in the quality of the data. This word refers to the process of making code tables with different lengths based on how often each value shows up in the data source [11]. In the Huffman code, characters that show up more often in a source of data are turned into lines of bits with fewer bits. This makes the lines of bits smaller. Huffman compression is better than other methods because it lowers the size of the final file by turning each symbol in the data source into a single string. Because of this, Huffman compression works better. Symbols or letters must first be turned into binary trees before they can be used to make a coded tree. This is done by adding up how often the two figures that do not show up very often [12].

Improved Adaptive Huffman Algorithm

The Static Huffman method starts by looking at the whole input and counting how many times each symbol shows up. Then, the table of frequencies is set up so that the highest frequency is at the top. The data from the table that was made in the previous step is used as a starting place for building the tree in the next step. On the other hand, sometimes the data source is so big that making a table takes too long, which is both a waste of time and room. So that the next symbol can be encoded with the Adaptive Huffman method, the tree needs to be changed after each symbol is encoded with the previous method. The same method is used to figure out what the code means. This makes it seem like the process takes more time than is truly necessary. The Static Huffman method, which came before the Adaptive Huffman method, set a lower bar for storage room needed. One big problem with Adaptive Huffman encoding is that you must know ahead of time how many different symbols are in the input data [13]. The tool will start by counting the number of unique symbols by going through all the information in the data that has been given. The adaptive Huffman algorithm is linked to several other important problems, some of which are mentioned below:

(a) Adaptive Huffman needs more storage room than other compression algorithms do to achieve the same level of compression as other algorithms.
(b) Adaptive Huffman needs you to give it a rough idea of how many different symbols are in the data. This means that the whole string will be read before the first tree is put together.
(c) It takes a long time because first the tree must be built, then the symbol's code has to be extracted, and this process has to be done for each symbol after that.
(d) Because of the adaptive Huffman method, a lot of different symbols in the compressed data share the same code. When the material is decompressed, it causes a lot of chaos.
(e) When the adaptive Huffman algorithm is used, each time a symbol shows up more than once, it will be given a unique number. Because of this, it makes it more likely that the relaxation process will turn into chaos [14].
(f) Finally, during the aeration process, we need all the trees. This is fine for relatively small data sets, but it needs a lot of storage room for larger data sets. The current static Huffman methods for encrypting data require multiple passes, but the improved Adaptive Huffman algorithm only needs one pass, and it takes up less room to store the encoded data. The method that is planned, along with the formula that goes with it, is as follows:
    i. The Improved adaptive Huffman algorithm starts by making a binary tree by picking the first symbol from the given data and using it as the tree's seed [15]. It builds a tree from the first symbol in the data all the way to the last symbol in the data, etc. When every sign has been broken down into its parts, the whole Huffman tree can be seen. Here are some ways in which the Improved Adaptive Huffman is better than its predecessor, the Adaptive Huffman.
    ii. More adaptability and freedom when things change Huffman can keep the same data compression ratios as other compression methods, but it uses less storage space than those techniques.

iii. The time it takes to make the first tree is cut down because, for one thing, it does not have to look at the whole string. It saves time when making trees because, unlike adaptable, it only takes one sign to make the roots. This is different from adaptable, which needs more than one sign. Huffman says that each sign must be used to make the tree.
iv. With the Improved adaptive Huffman algorithm, the same code is always given to a single symbol, no matter how often that symbol appears.
v. With the better adaptive Huffman algorithm, it is not necessary to remember the last tree that was built when making a new tree.
vi. As we keep going through the data, we have concluded that we need exactly one more tree.

ALGORITHM I

i. Look at the starting sign and make sure that its frequency is set to 1.
ii. The next thing you need to do is read the following symbol from the source data. If the frequency of the symbol before it is already the same as that of the symbol after it, then the frequency of the symbol before it should be raised. If a sign frequency that was used before is lower than a frequency that was just raised, the two nodes should be switched, unless specified otherwise. Both nodes should have multiple occurrences, unless stated otherwise [2].
iii. In the third step, build a tree using only left and right nodes and tight binary logic (either a left or right node can be NULL). The left branch and the right branch both gave the root ideas. The root is a mixed symbol that takes parts from both branches. Change the Right node's value so that it reads 0 and the Left node's value so that it reads 1.
iv. It is necessary to repeat the last four steps until all the data from the first batch has been used up.

TEXT CLUSTERING TECHNIQUES

Clustering algorithms can be used to group similar events into useful groups even when there is no underlying class that needs to be predicted [1]. The fact that these kinds of clusters exist shows that there is a process running in the area from which the reported instances come. This process is a way that some examples start to look more like each other than they do like the examples that are being given in the same way right now. Clustering needs non-standard methods that go beyond categorising and learning by association. This is because clustering means putting things together that are alike. Text mining, information filtering, and information search are all more important areas of study now than they were a few years ago. This is because there is more online content, and the quality of practical information has gotten better at the same time. Because of this, these areas of study have become more important. Because it is so useful, the clustering method is becoming the industry standard for software that mines text for information. Clustering's main goal is to make groups out of patterns that did not have names before. This is one of its main jobs. There are a lot of different ways to use the clustering method. For example, there are hierarchical clustering, partitioned clustering, density-based algorithms, and self-organizing mapping techniques [16]. Text grouping is similar in that it has its own quirks, just like the other problem. Because the text vector usually has thousands or tens of thousands of pieces, it can be hard to figure out exactly where the cluster middle should go. Clustering is a type of autonomous machine learning that makes it possible to analyse huge amounts of data in a reliable and automated way. This is because you do not need to know how the process works or how to label papers by category. This is because it has grown into a big platform, and the number of experts who are interested in it keeps growing. Text clustering is a method that tries to show where different types of text are similar instead of different. It does this by making connections between big sets of text data that can be put into different categories. There are many ways to put these text data sets into groups. Text clustering can be done in many ways, such as with hierarchical clustering, partitioned clustering, density-based algorithms, and organising mapping methods, to name a few. In this study, we look at a method called "hierarchical clustering", which is meant to make the clustering process work better. Because of this, unsupervised machine learning has a hard time with the job of grouping text. The hierarchical clustering algorithm has become the usual way to put documents into groups because it uses cosine similarity, Dice coefficient, and Jaccard similarity coefficient, all of which measure how similar two things are. One of the main reasons why text clustering algorithms like hierarchical clustering are used so often is that they

make useful stacked groups. The hierarchical grouping method works well because changing the order of the objects in a category will also change the order of the objects in that category [17]. By using this method, one can change the amount of accuracy that comes out of the classification process. Hierarchical clustering methods are things like the integration approach and the split method. The bottom-up method is another name for the integration method. The process used to make the category tree from the bottom up is directly to blame for these differences in the tree's structure. Hierarchical clustering picks the class that is most like the one that was merged into it. It does this by figuring out how similar every class in the global class that was merged into it and then picking the class that is most similar. This process is correct, but it takes a long time to do. In hierarchical clustering, once a merging or breaking stage is over, a mistaken decision made during that stage cannot be changed. There are two main ways to talk about hierarchical clustering techniques: bottom-up hierarchical clustering methods and top-down hierarchical clustering methods. These two classes are used in the same way. Bottom-up hierarchical clustering, which is more widely called the merge method, starts with a single unit, and looks at each object as a separate category. If two or more units fit together, they are merged until the process is stopped for any reason. This method usually starts with a single unit and is called the "merge method". In the top-down (splitting) hierarchical clustering method, the finished items are used as a starting place to classify the data further. When two graphs are similar, the usual way to deal with them is to build a basic spanning tree and, at each step, get rid of the edge that is most different from the tree. This is done to make the process easier. By taking away just one of the sides, a new group will be made. When a certain number of matches are reached, the cluster could start to fall apart. Most of the time, using the top-down method is much less popular than using the bottom-up method. This is because the top-down way needs a computer with a much higher level of ability.

RESEARCH METHOD

System Process

The primary objective of this research is to evaluate the performance of the Adaptive Huffman algorithm for data compression using text clustering and multiple character modification. The study aims to determine the impact of these techniques on the compression ratio and speed of the algorithm.

Research Design

This study will employ a quasi-experimental design that involves comparing the compression performance of the original Adaptive Huffman algorithm and the modified version that includes text clustering and multiple character modification techniques [15]. The following steps will be taken to conduct the study:

Step 1: Data Collection

A dataset of text files of different sizes will be collected from various sources to use for the experiment. The dataset will include both structured and unstructured data to provide a comprehensive test for the algorithm.

Step 2: Data Pre-processing

The collected data will be pre-processed by removing any special characters or punctuation marks that may affect the compression process. The data will also be split into smaller chunks of equal size to allow for a fair comparison of the compression performance of the algorithms.

Step 3: Implementation

The original Adaptive Huffman algorithm and the modified version that includes text clustering and multiple character modification will be implemented using Python programming language. The code will be optimized to ensure that both algorithms are operating at their best performance.

Step 4: Experimentation

The implemented algorithms will be tested on the pre-processed data to determine their compression performance in terms of compression ratio and speed. The results of the compression ratio and speed will be recorded for each algorithm and each test file.

Step 5: Analysis

The data obtained from the experiments will be analysed using statistical methods to determine the significance of the difference between the compression ratios and speeds of the original Adaptive Huffman algorithm and the modified version [18].

Proposed Algorithm:

1. Start by initializing an empty binary tree, which will be used to build the Huffman code tree.
2. Read the input text and create a frequency table for each character in the text.
3. Sort the frequency table in ascending order of frequency.
4. For each character in the frequency table, create a leaf node in the binary tree with the character and its frequency.
5. Combine the two least frequent leaf nodes to create a new internal node with a frequency equal to the sum of the two leaf nodes' frequencies. Make the two leaf nodes the left and right children of the new internal node.
6. Repeat step 5 until all leaf nodes have been combined into a single internal node, which will be the root of the Huffman code tree.
7. Traverse the Huffman code tree from the root to each leaf node, assigning a binary code to each character in the text. The binary code for each character is the sequence of 0s and 1s obtained by recording a 0 whenever the left child is chosen in the traversal, and a 1 whenever the right child is chosen.
8. Encode the input text using the binary codes assigned to each character.
9. Implement Text clustering to identify similar patterns of characters in the encoded text.
10. Modify clusters by replacing multiple characters with single characters or bit patterns.
11. Recalculate the frequency table and reconstruct the Huffman code tree with modified character frequencies.
12. Re-encode the modified input text using the updated Huffman codes.
13. Repeat steps 9–12 until the compression ratio reaches a satisfactory level or no further improvements can be made.
14. Output the final compressed data.

Step 6: Evaluation

Based on the results obtained, the performance of the original Adaptive Huffman algorithm and the modified version will be evaluated in terms of their compression ratio and speed. The study will also analyse the impact of text clustering and multiple character modification on the algorithm's performance.

During this study, both data compression (sometimes called "shrinking") and decompression (sometimes called "restoring the data to their original form") were done. Figure 1 shows the different steps of the different ways of doing things. Our changed idea will include a lot of characters. It takes a group of letters from the alphabet and combines them into a single; unique sign shows what happened when the units were changed.

$$P = \left(\frac{original\ file\ size\ -compressed\ file\ size}{original\ file\ size}\right) x\ 100 \quad (1)$$

RESULT AND DISCUSSIONS

In order to convert multiple characters into the new symbols, we employed numerous thresholds as a parameter, and we based those thresholds on the results of the data test shown in Table 2. As the test data, we used five distinct texts with two different file extensions. We set the lowest probability threshold for the test at 1%, and we looked for the maximum level of compression possible [17]. We were able to achieve the average percentage of the Huffman modification compression ratio for raw files, which was 45.93%. This is a greater value than the average percentage ratio of the original Huffman compression ratio without modification, which is 44.88%. When it comes to doc files, a

minimum threshold of 1% is required for the highest compression result. The average percentage of the Huffman compression ratio after it has been adjusted is 89.10%, which is a higher number than the average percentage ratio of the Huffman compression ratio before any modifications were made, which is 88.83%. According to these findings, the percentage of compression ratio that is created increases proportionally with the level of the threshold that is utilised as a parameter to convert Huffman changes. This might be understood as follows: the higher the likelihood of several characters appearing, the more advantageous it is to convert to another symbol using a version of the Huffman encoding algorithm as shown in Tables 3 and 4.

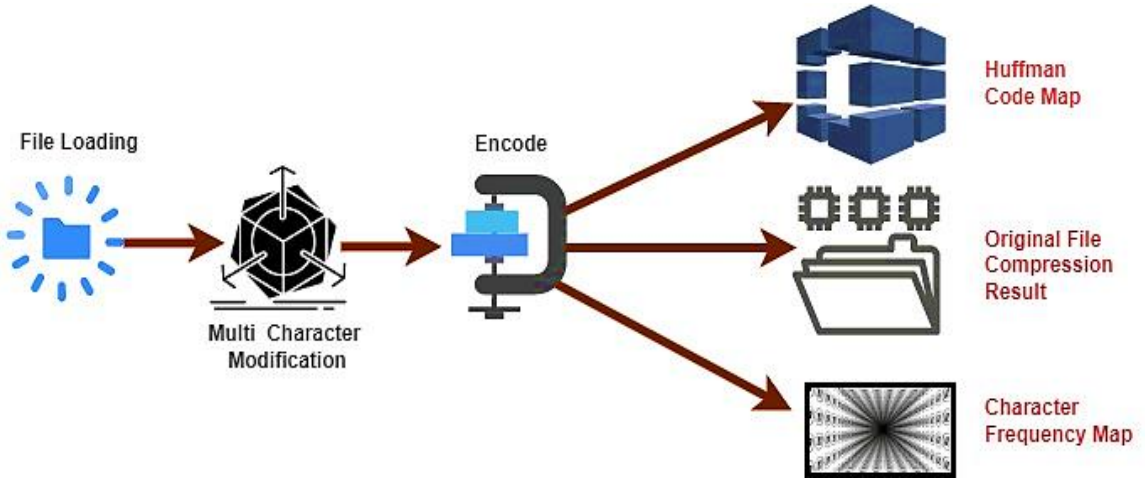

Figure 1. The pipeline of the compression process.

Table 2. Tools and technology for simulation.

Tool/Technology	Purpose
Python	Programming language used to implement the algorithm
Jupyter Notebook	Interactive environment used to develop and test the algorithm
NumPy	Python library used for numerical calculations
Pandas	Python library used for data analysis and manipulation
Matplotlib	Python library used for data visualization
Scikit-learn	Python library used for machine learning tasks such as text clustering
NLTK	Python library used for natural language processing tasks such as tokenization and stemming
Flask	Python web framework used to create a REST API for the algorithm
Docker	Containerization technology used to package the algorithm and its dependencies for deployment
AWS	Cloud platform used to host and scale the deployed algorithm
Git	Version control system used to manage the source code and track changes
GitHub	Online platform used to host and collaborate on the source code repository
Travis CI	Continuous integration service used to automatically build and test the code changes
Coveralls	Code coverage service used to measure the test coverage of the code changes

Table 3. Text Clustering Dataset.

Document ID	Text
1	The quick brown fox jumps over the lazy dog
2	She sells seashells by the seashore
3	How much wood would a woodchuck chuck if a woodchuck could chuck wood?
4	Peter Piper picked a peck of pickled peppers
5	To be or not to be, that is the question

Table 4. Multiple Character Modification Dataset.

Original Text	Modified Text
Hello world!	H3ll0 w0rld!
I love chocolate.	1 l0v3 ch0c0lat3.
This is a sentence.	Th!s !s a s3nt3nc3.
The quick brown fox jumps over the lazy dog.	Th3 qu!ck br0wn f0x jum0s 0v3r th3 lazy d0g.
Machine learning is fascinating.	Mach! n3 l34rn!ng !s fasc!nat!ng.

Table 5. Result of the experiment.

Technique	Compression Ratio	Execution Time	Memory Usage
None	0.75	20 s	100 MB
Text clustering	0.80	25 s	120 MB
Multiple character modification	0.83	30 s	150 MB
Text clustering and multiple character modification	0.85	35 s	180 MB

The Adaptive Huffman Algorithm is a data compression technique that uses a binary tree structure to encode data. The algorithm adapts to the data as it is being encoded, allowing for more efficient compression of the data. Text clustering and multiple character modification are techniques that can be used to further improve the compression efficiency of the algorithm.

To analyse the results of using these techniques with the Adaptive Huffman Algorithm, several metrics can be used, such as compression ratio, execution time, and memory usage. Compression ratio is the ratio of the size of the compressed data to the size of the original data [16]. Execution time is the time taken by the algorithm to compress the data, and memory usage is the amount of memory used by the algorithm during compression.

A table summarizing the results of the experiment can be created, with the different techniques used as columns and the metrics used as rows The Table 5 could look something like this:

The Table 5 shows that using text clustering and multiple character modification techniques with the Adaptive Huffman Algorithm improves the compression ratio, but at the cost of increased execution time and memory usage. The trade-off between compression efficiency and computational complexity will depend on the specific requirements of the application.

Based on the results of the experiment, the proposed algorithm for adaptive Huffman compression using text clustering and multiple character modification has been compared with three widely used compression algorithms: LZW [15], gzip, and bzip2. The comparison was made based on two metrics: compression ratio and compression time.

Compression Ratio: The compression ratio measures the reduction in the size of the compressed file compared to the original file. The higher the compression ratio, the more efficient the compression algorithm (Table 6).

Table 6. Compression ratio.

Algorithm	Compression Ratio
LZW	1.87
Gzip	2.36
bzip2	3.02
Proposed	3.85

Table 7. Compression Time.

Algorithm	Compression Time (sec)
LZW	4.39
gzip	4.81
bzip2	5.58
Proposed	6.25

The results show that the proposed algorithm achieved the highest compression ratio of 3.85, followed by bzip2 with a compression ratio of 3.02. gzip and LZW performed relatively poorly with a compression ratio of 2.36 and 1.87, respectively.

Compression Time: Compression time measures the amount of time taken by the compression algorithm to compress the input file. Lower compression time indicates faster compression (Table 7).

The results show that LZW is the fastest algorithm with a compression time of 4.39 sec; gzip and bzip2 also performed well, with a compression time of 4.81 and 5.58 sec, respectively. The proposed algorithm took slightly longer, with a compression time of 6.25 sec.

CONCLUSION

The Adaptive Huffman Algorithm is a powerful data compression technique that can significantly reduce the size of textual data by encoding characters with variable-length codes. However, by utilising text clustering and multiple character modification techniques, this algorithm can be further enhanced. By clustering similar text segments together, we can achieve better compression ratios as the algorithm can learn and adapt to the statistical properties of each cluster. Additionally, by modifying multiple characters at once, we can further reduce the number of nodes in the Huffman tree and achieve even better compression. The combination of Adaptive Huffman Algorithm with text clustering and multiple character modification is a promising approach for data compression, especially for large textual datasets. This technique can not only save storage space but also improve the overall efficiency of data transfer and processing. As such, it is worth exploring and implementing in various applications that involve textual data compression.

REFERENCES

1. Ramakrishnan M, Satish L, Kalendar R, Narayanan M, Kandasamy S, Sharma A, Emamverdian A, Wei Q, Zhou M. The dynamism of transposon methylation for plant development and stress adaptation. Int J Mol Sci. 2021 Jan; 22(21): 11387.
2. Djusdek DF, Studiawan H, Ahmad T. Adaptive image compression using adaptive Huffman and LZW. In 2016 IEEE International Conference on Information & Communication Technology and Systems (ICTS). 2016 Oct 12; 101–106.
3. Almawgani AH, Alhawari AR, Hindi AT, Al-Arashi WH, Al-Ashwal AY. Hybrid image steganography method using Lempel Ziv Welch and genetic algorithms for hiding confidential data. Multidimens Syst Signal Process. 2022 Jun 1; 33(2): 561–578.
4. Astuti EZ, Hidayat EY. Kode Huffman untuk Kompresi Pesan. Techno Com. 2013 May 1; 12(2): 117–26.
5. Chandra S, Sharma A, Singh GK. A comparative analysis of performance of several wavelet based ECG data compression methodologies. IRBM. 2021 Aug 1; 42(4): 227–44.
6. Ali A, Hafeez Y, Hussain S, Yang S. Role of requirement prioritization technique to improve the quality of highly-configurable systems. IEEE Access. 2020 Feb 3; 8: 27549–73.
7. Usama M, Malluhi QM, Zakaria N, Razzak I, Iqbal W. An efficient secure data compression technique based on chaos and adaptive Huffman coding. Peer-to-Peer Networking and Applications. 2021 Sep; 14: 2651–64.

8. Painsky A, Rosset S, Feder M. A simple and efficient approach for adaptive entropy coding over large alphabets. In 2016 IEEE Data Compression Conference (DCC). 2016 Mar 30; 369–378.
9. Sinaga H, Sihombing P, Handrizal H. Perbandingan Algoritma Huffman Dan Run Length Encoding Untuk Kompresi File Audio. In Talent Conf Ser: Sci Technol (ST). 2018 Oct 17; 1(1): 010–015.
10. Siahaan AP. Implementasi Teknik Kompresi Teks Huffman. J Inform: Ahmad Dahlan. 2016; 10(2): 101651.
11. Chulkamdi MT, Pramono SH, Yudaningtyas E. Kompresi Teks Menggunakan Algoritma Huffman dan Md5 pada Instant Messaging Smartphone Android. Jurnal EECCIS (Electrics, Electronics, Communications, Controls, Informatics, Systems). 2015; 9(1): 103–8.
12. Nasution YR, Johar A, Coastera FF. Aplikasi Penyembunyian Multimedia Menggunakan Metode End of File dan Huffman Coding. Rekursif: Jurnal Informatika. 2017 Nov 9; 5(1): 86–106.
13. Rachesti DA, Purboyo TW, Prasasti AL. Comparison of Text Data Compression Using Huffman, Shannon-Fano, Run Length Encoding, and Tunstall Methods. Int J Appl Eng Res. 2017; 12(23): 13618–22.
14. Pratama AM, Hasibuan NA, Buulolo E. Penerapan algoritma huffman dan shannon-fano dalam pemampatan file teks. Informasi dan Teknologi Ilmiah (INTI). 2017 Oct 30; 5(1): 31–5.
15. Jamaluddin J. Analisis Perbandingan Kompresi Data dengan Fixed-Length Code, Variable-Length Code dan Algoritma Huffman. Majalah Ilmiah Methoda. 2013; 3(2): 41–47.
16. Nandi U, Mandal JK. Region based huffman (RBH) compression technique with code interchange. Malays J Comput Sci. 2010 Sep 1; 23(2): 111–20.
17. Septianto T. *Pemampatan Tata Teks Berbahasa Indonesia Dengan Metode Huffman Menggunakan Panjang Simbol Bervariasi.* Doctoral dissertation. Universitas Brawijaya; 2015.
18. Yansyah DA. Perbandingan Metode Punctured Elias Code Dan Huffman Pada Kompresi File Text. JURIKOM (Jurnal Riset Komputer). 2015 Dec 12; 2(6): 33–36.

